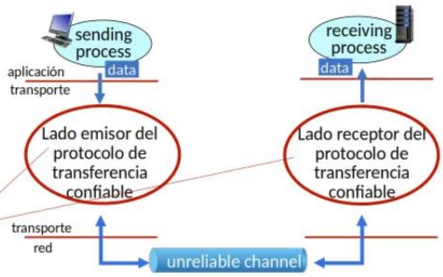
Principios de transferencia confiable

abstracción de servicio confiable

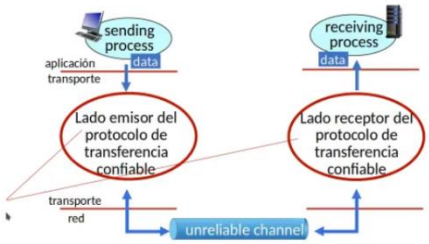
La complejidad del protocolo de transferencia confiable dependerá ampliamente de las características del canal no confiable (¿datos perdidos, reordenar datos alterados, desordenados?)



Implementación del servicio confiable

El emisor y el receptor desconocen el estado del otro, e.g., *¿se recibió un paquete?*

- a menos que se comunique a través de un mensaje



Implementación del servicio confiable

rdt2.0: canal propenso a errores

- El canal puede invertir los bits en los paquetes
  - La función de *checksum* sirve para detectar errores
- ¿Cómo recuperarse de dichos errores?

Durante una conversación, ¿cómo se "recuperan" las personas de las palabras no comprendidas?

rdt3.0: canales con errores y pérdidas

Suposición: *el canal puede perder paquetes* (datos, ACKs)

- El checksum, los #s de secuencia, los ACKs y las retransmisiones serán útiles ... pero no lo suficiente

Q: ¿De qué forma manejan las personas palabras "perdidas" en una conversación?

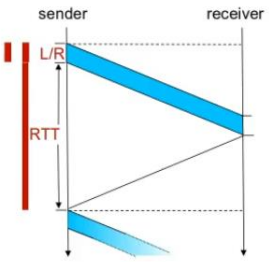
Performance of rdt3.0 (stop-and-wait)

- U_{emisor}: *utilización* – fracción de tiempo en el que el emisor está ocupado
- ejemplo: enlace de 1 Gb/s, d_{prop} = 15 ms, paquete de 8000 bits
  - tiempo para poner el paquete en el canal:

$$d_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ \mu sec$$

Operación de stop & wait

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

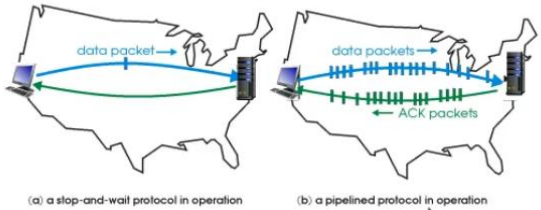


- ¡Stop & wait es pésimo!
- El protocolo subutiliza considerablemente el canal

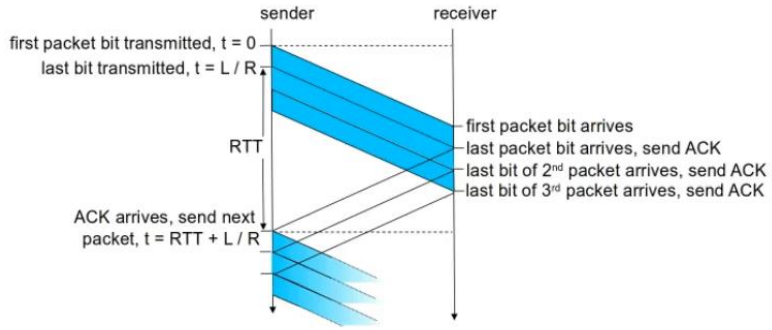
Operación de protocolos en *pipeline*

pipelining: el emisor puede enviar varios paquetes sin recibir acuse de recibo

- El rango de los números de secuencia debe aumentarse
- Se requiere de almacenamiento en la fuente y/o en el receptor



Pipelining: increased utilization



$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

Go-Back-N: emisor

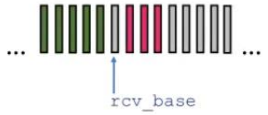
- emisor: "ventana" de hasta *N* paquetes consecutivos sin recibir ACK
  - # de secuencia de k-bits en el encabezado del paquete

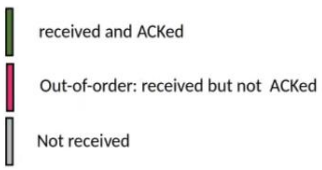
- *ACKs acumulativos:* ACK(*n*) acusa de recibo los paquetes hasta el *n*
  - al recibir ACK(*n*): mover la ventana para comenzar en *n+1*
- timer para el paquete enviado más "viejo"
- *timeout(n)*: retransmite el paquete *n* y todos los demás con mayor número de secuencia en la ventana

Go-Back-N: receptor

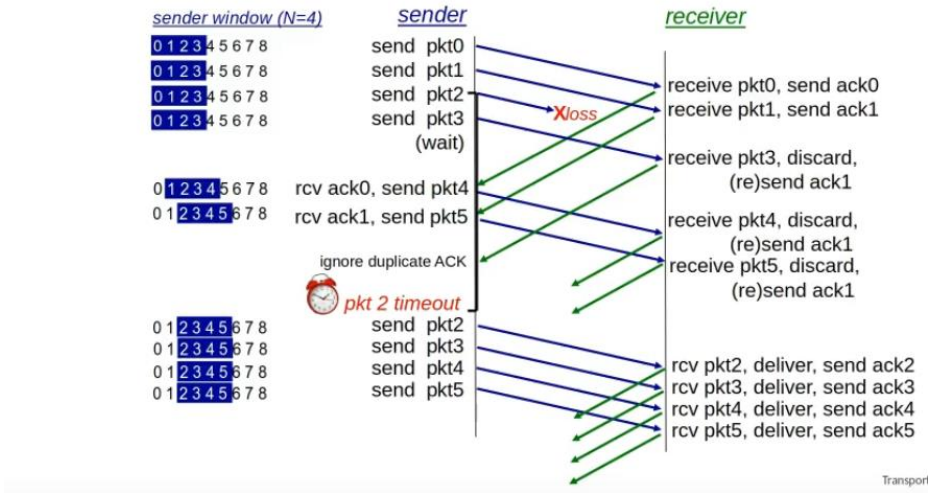
- sólo ACK: siempre envía ACK para el paquete bien recibido hasta ese momento con el número de secuencia *en orden* más alto
  - puede generar ACKs duplicados
  - sólo necesita recordar `rcv_base`
- al recibir un paquete desordenado:
  - puede descartarlo (no almacenarlo): decisión de implementación
  - reenviar ACK para el paquete con el mayor número de secuencia ordenado

Visión del receptor del espacio de # seq:





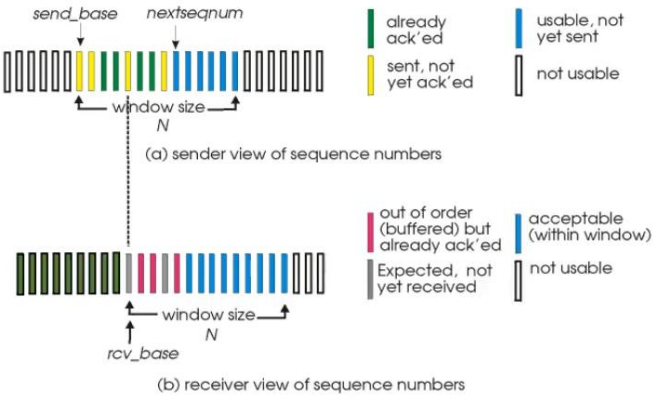
Go-Back-*N* en acción



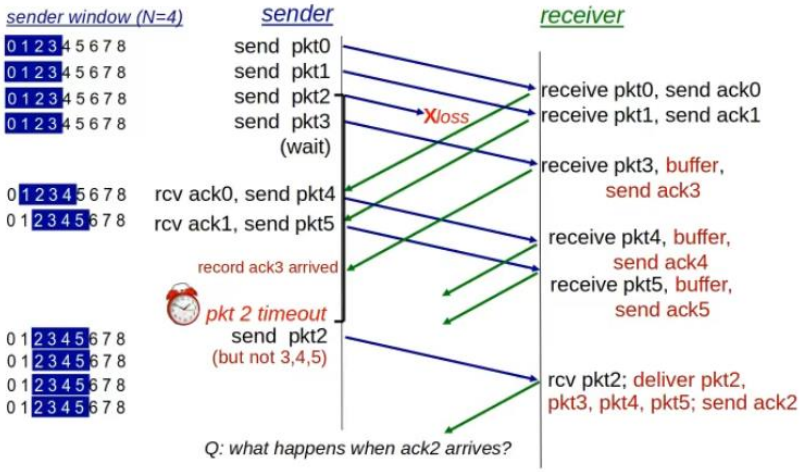
Selective repeat

- El receptor acusa de recibo *individualmente* todos los paquetes correctamente recibidos
  - Almacena los paquetes, conforme lo necesita, para una entrega eventual ordenada a la capa superior
- El emisor retransmite individualmente los paquetes para los cuales no recibió ACK
  - gestiona un timer para cada paquete sin ACK
- ventana del emisor
  - *N* números de secuencia consecutivos
  - limita los números de secuencia de paquetes enviados y sin ACK

Selective repeat: ventanas del emisor y receptor



Selective Repeat in action

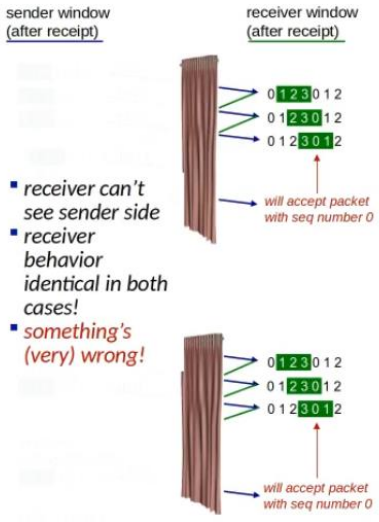


Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?



- *receiver can't see sender side*
- *receiver behavior identical in both cases!*
- *something's (very) wrong!*

7-oct

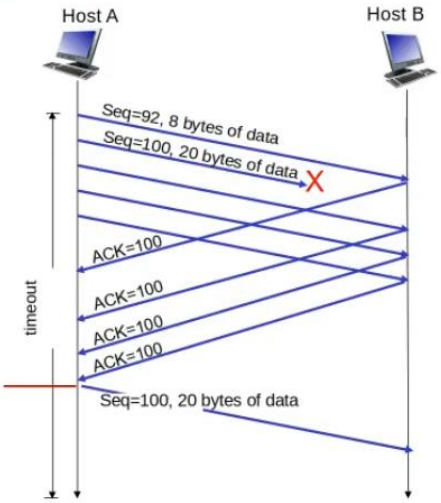
TCP fast retransmit

TCP fast retransmit

Si el emisor recibe tres ACKs adicionales para los mismos datos ("triple duplicate ACKs"), reenviar el segmento sin ACK con el menor #seq

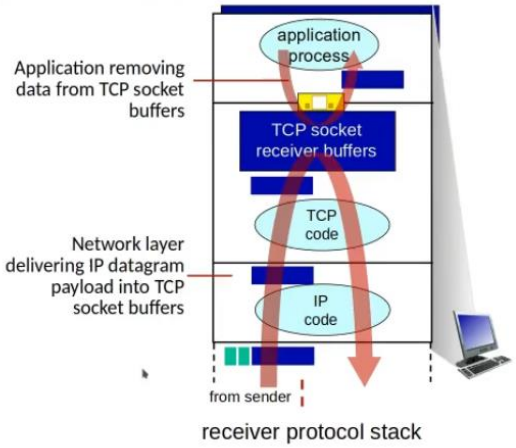
- Muy probable que el segmento sin ACK se perdió, así que no hay que esperar el timeout

Recibir tres ACKs duplicados indica 3 segmentos recibidos después de uno perdido, ¡así que se decide retransmitir!



Control de flujo en TCP

Q: ¿Qué pasa si la capa de red entrega los datos más rápido de lo que la capa de aplicación lee de los buffers de los *sockets*?



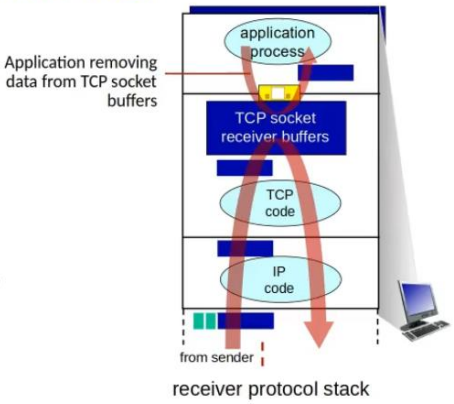
El control de flujo consiste que el receptor envía sus segmentos a una velocidad que el receptor pueda tolerar no más rápido es una regulación de la transmisión.

Control de flujo en TCP

Q: ¿Qué pasa si la capa de red entrega los datos más rápido que lo que la capa de aplicación lee de los buffers de los sockets?

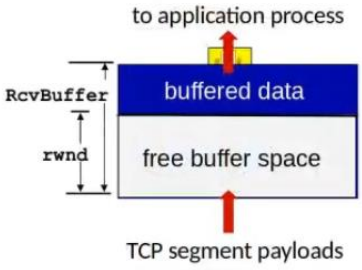
flow control

el receptor controla al emisor, así que el emisor no saturará el buffer del receptor transmitiendo demasiado, muy rápido



Control de flujo en TCP

- El receptor TCP "anuncia" en el campo **`rwnd`** el espacio libre en el buffer
  - El tamaño de **`RcvBuffer`** se configura con las opciones del socket (valor típico por default es 4096 bytes)
  - muchos sistemas operativos autoadjustan **`RcvBuffer`**
- el emisor limita al valor recibido de **`rwnd`** la cantidad de datos unACKed ("in-flight")
- garantiza que el receptor no desbordará



TCP receiver-side buffering

Control de flujo: ejemplo

- Las estaciones A y B están conectadas directamente con un enlace de 100 Mb/s. Hay una conexión TCP entre las dos estaciones y la estación A está enviando a B un archivo enorme sobre dicha conexión. La estación A puede enviar datos de aplicación sobre el enlace a 50 Mb/s, pero la estación B puede leer de su buffer de recepción TCP a una velocidad máxima de 20 Mb/s. Describa el efecto del control de flujo de TCP.

  - La Estación A envía datos más rápido que lo que B puede recibir
  - El buffer de recepción se llena a una velocidad de 50 – 20 = 30 Mb/s
  - Cuando se llena el buffer, poniendo rwnd = 0, B le avisa a A que deje de transmitir
  - A se detiene y retoma al recibir rwnd > 0. A reiniciará y se detendrá con base en los valores de rwnd que reciba de B.
  - Finalmente, esto resultará en una velocidad promedio de recepción en B cercana a los 20 Mb/s

Principios de control de congestión

Congestión:

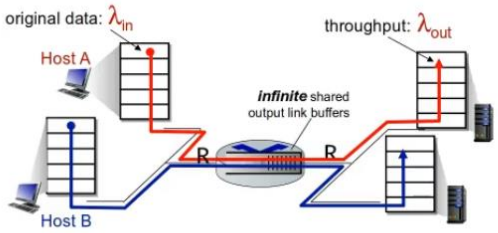
- informalmente: "demasiadas fuentes envían demasiados datos muy rápido, para que la *red* pueda manejarlos"
- síntomas:
  - grandes retardos (formación de filas en los ruteadores)
  - pérdida de paquetes (desbordamiento de las filas)
    - ¡diferente del control de flujo!
    - ¡un problema en el top 10!

congestion control: too many senders, sending too fast

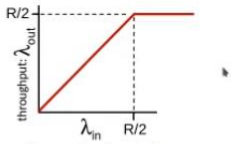
flow control: one sender too fast for one receiver

Causas/costos de la congestión: escenario 1

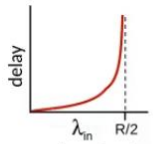
Escenario más simple:
- un ruteador, buffers infinitos
- input, output link capacity: *R*
- dos flujos
- no se necesitan retransmisiones



Q: ¿qué pasa conforme la tasa de llegada λ_{in} se acerca a *R*/2?



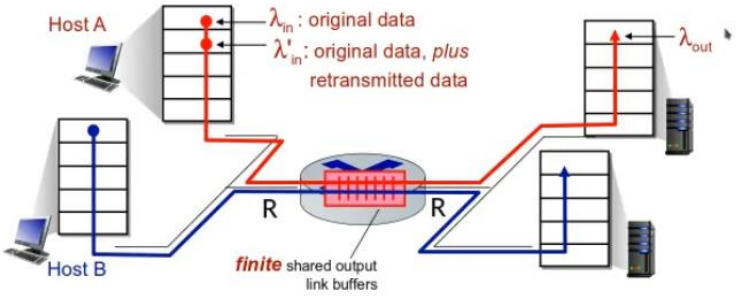
throughput máximo por conexión: *R*/2



grandes retardos conforme λ_{in} se aproxima a la capacidad

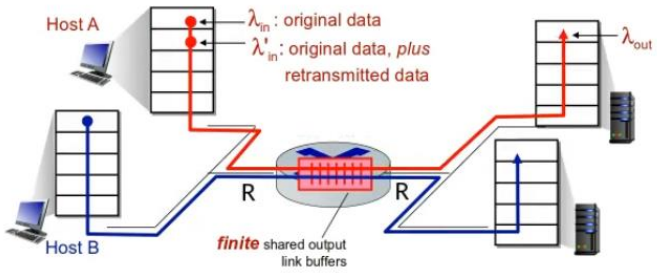
Causas/costos de la congestión: escenario 2

- un ruteador, buffers *finitos*
- el emisor retransmite paquetes perdidos y expirados
  - entrada de la capa de aplicación = salida de esa capa: $\lambda_{in} = \lambda_{out}$
  - la capa de transporte incluye *retransmisiones* : $\lambda'_{in} \geq \lambda_{in}$



Causas/costos de la congestión: escenario 2

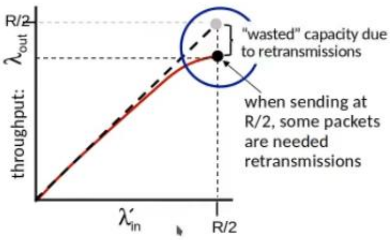
Idealización: conocimiento perfecto
- el emisor transmite únicamente cuando el ruteador tiene buffers disponibles

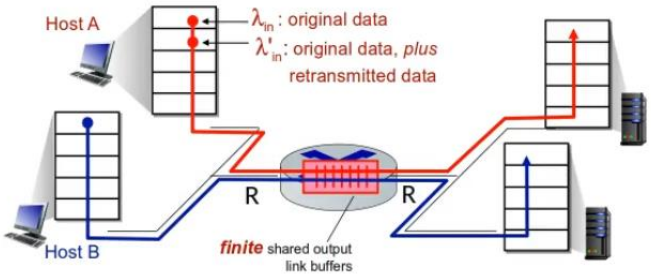


Causas/costos de la congestión: escenario 2

Idealización: *algún* conocimiento perfecto

- los paquetes se pueden perder (en el ruteador) porque la fila se satura
- el emisor sabe cuando el paquete se ha perdido: sólo reenvía si *sabe* que se perdió

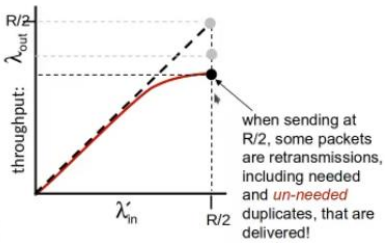


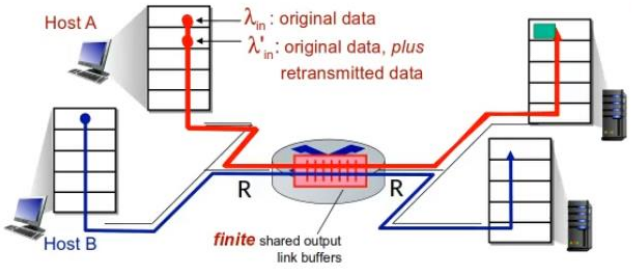


Causas/costos de la congestión: escenario 2

Escenario realista: *duplicados innecesarios*

- los paquetes pueden perderse en el ruteador porque encuentran llena la fila – requiriendo retransmisiones
- el emisor puede incurrir en *timeouts* prematuros, enviando *dos* copias, *ambas* se entregan

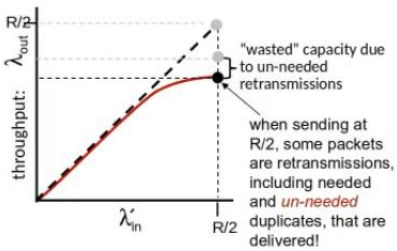




Causas/costos de la congestión: escenario 2

Escenario realista: *duplicados innecesarios*

- los paquetes pueden perderse en el ruteador porque encuentran llena la fila – requiriendo retransmisiones
- el emisor puede incurrir en *timeouts* prematuros, enviando *dos* copias, *ambas* se entregan



"costos" de la congestión:

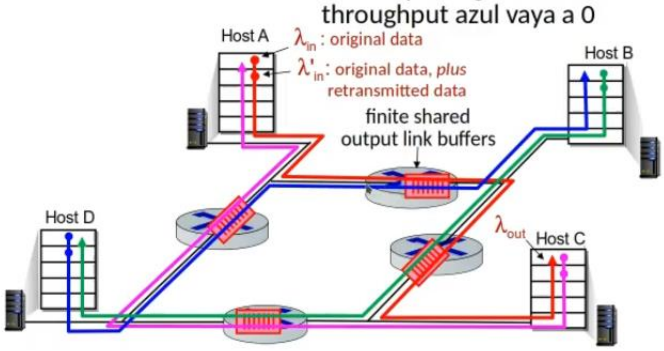
- más trabajo (retransmisión) para un throughput dado del receptor
- retransmisiones innecesarias: el enlace lleva varias copias de un paquete
  - Lo cual disminuye el throughput máximo alcanzable

Causas/costos de la congestión: escenario 3

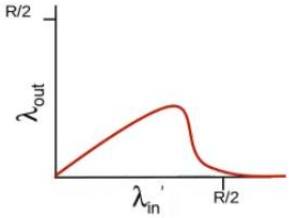
- *cuatro* emisores
- caminos *multi-hop*
- timeout/retransmit

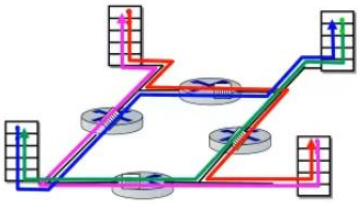
Q: ¿qué pasa conforme λ_{in} y λ_{in}' aumentan?

A: conforme λ_{in}' roja aumenta, todos los paquetes azules que llegan a la fila de arriba, hacen que el throughput azul vaya a 0



Causas/costos de la congestión: escenario 3



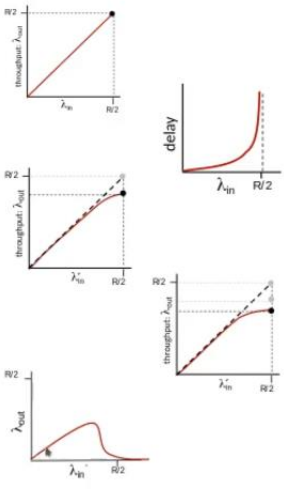


otro "costo" de la congestión:

- cuando se pierde un paquete, ¡se pierden la capacidad de transmisión y almacenamiento usadas para el paquete!

Causas/costos de la congestión: revelaciones

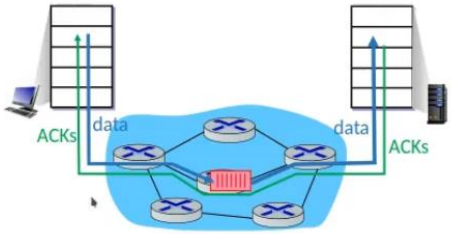
- el throughput nunca puede exceder a la capacidad
- el retardo aumenta conforme se aproxima a la capacidad
- pérdidas/retransmisión reducen el throughput efectivo
- los duplicados innecesarios reducen aún más el throughput efectivo
- se pierde la capacidad de transmisión / el buffering para los paquetes de regreso (downstream)



Enfoques para el control de congestión

Control de congestión end-to-end:

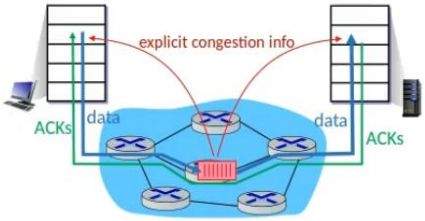
- Sin retroalimentación específica de la red
- Se *infiere* la congestión a partir del retardo y pérdidas observados
- enfoque que sigue TCP



Enfoques para el control de congestión

Control de congestión asistido por la red:

- Los ruteadores ofrecen retroalimentación *directa* a las estaciones emisora/receptora con los flujos que atraviesan el ruteador congestionado
- se puede indicar un nivel de congestión o configurar explícitamente una velocidad



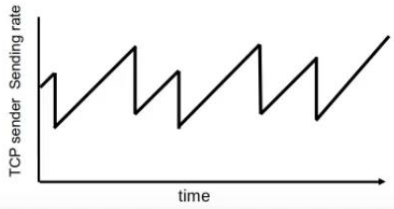
- Protocolos TCP ECN, ATM, DECbit

Control de congestión en TCP: AIMD

- *enfoque:* los emisores pueden aumentar su tasa de transmisión hasta que ocurran pérdidas (congestion) occurs, entonces disminuirla

Additive Increase
aumentar la tasa de transmisión en un MSS cada RTT hasta detectar pérdida

Multiplicative Decrease
dividir por dos la tasa de transmisión luego de cada pérdida



Diente de sierra de AIMD: *para inferir* el ancho de banda

TCP AIMD: más

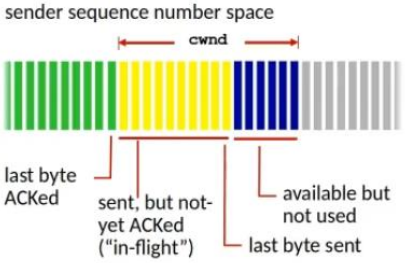
Multiplicative decrease detail: la tasa de transmisión se

- divide a la mitad luego de detectar pérdidas por un triple ACK duplicado (TCP Reno)
- reduce a 1 MSS (maximum segment size) cuando se detecta pérdida por *timeout* (TCP Tahoe)

¿Por qué AIMD?

- AIMD – es un algoritmo distribuido, asíncrono – ha mostrado que:
  - ¡optimiza las velocidades de flujos congestionados en toda la red!
  - tiene propiedades deseables de estabilidad

Control de congestión en TCP: detalles



Comportamiento de transmisión:

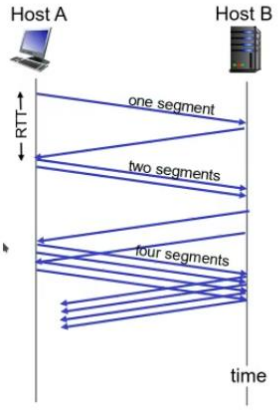
- *aprox:* enviar `cwnd` bytes, esperar RTT luego de los ACKS, entonces enviar más bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- El emisor TCP limita la transmisión: `LastByteSent- LastByteAcked ≤ cwnd`
- `cwnd` se ajusta dinámicamente en respuesta a la congestión observada (implementando control de congestión TCP)

TCP slow start

- Cuando inicia la conexión, aumentar exponencialmente la velocidad hasta la primera pérdida:
  - inicialmente **`cwnd`** = 1 MSS
  - duplicar **`cwnd`** cada RTT
  - se hace aumentando **`cwnd`** luego de recibir cada ACK
- *resumen:* la vel. inicial es lenta, pero crece exponencialmente rápido



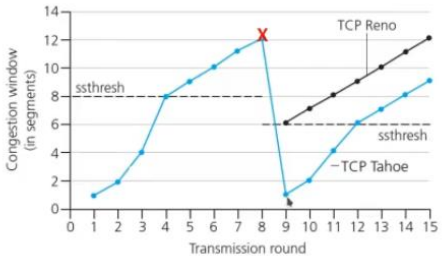
TCP: from slow start to congestion avoidance

Q: ¿cuándo se debe pasar de crecimiento exponencial a lineal?

A: cuando **`cwnd`** llega a 1/2 de su valor antes del timeout.

Implementación:

- variable **`ssthresh`**
- luego de una pérdida, **`ssthresh`** se pone a 1/2 del valor de **`cwnd`** que había justo antes del evento



9-oct

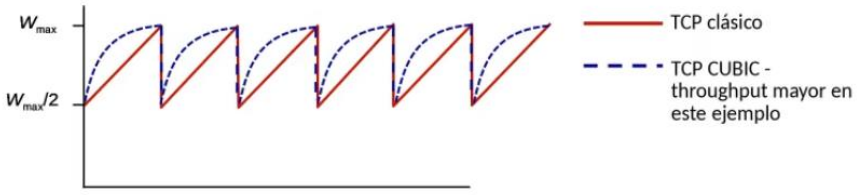
Problema 40, cap. 3 Kurose & Ross

h) ¿Durante cuál vuelta de transmisión se envía el segmento 70?

Vuelta	cwnd	Segmentos
1	1	1
2	2	2, 3
3	4	4, 5, 6, 7
4	8	[8, 15]
5	16	[16, 31]
6	32	[32, 63]
7	**33**	**[64, 76]**

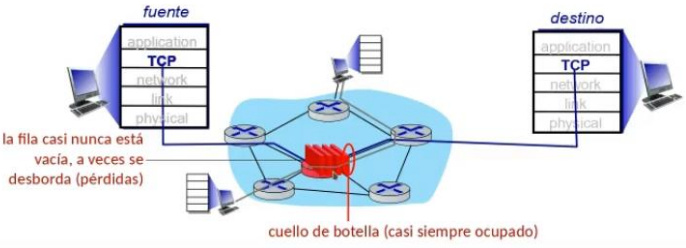
TCP CUBIC

- ¿Hay algo mejor que AIMD para inferir el ancho de banda disponible?
- Observación/intuición:
  - W_{max}: tasa de envío a la cual se detectó pérdida por congestión
  - El estado de congestión del cuello de botella probablemente no ha cambiado tanto
  - Luego de disminuir la ventana a la mitad, inicialmente llegar a W_{max} *más rápido*, pero entonces acercarse a W_{max} más *lentamente*



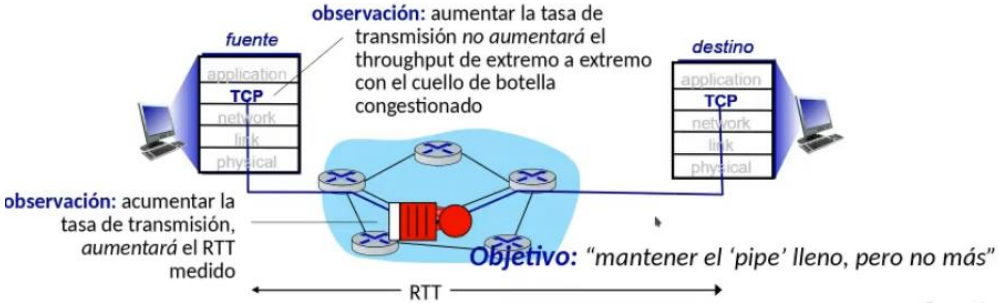
TCP y el cuello de botella congestionado

- TCP (clásico, CUBIC) aumentan su tasa de envío hasta que ocurran pérdidas en la salida de algún ruteador: el *cuello de botella*



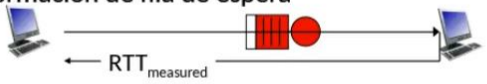
TCP y el cuello de botella congestionado

- TCP (clásico, CUBIC) aumentan su tasa de envío hasta que ocurran pérdidas en la salida de algún ruteador: el *cuello de botella*
- comprender la congestión: útil para enfocarse en el enlace congestionado



Control de congestión basado en retardo

Mantener el 'pipe' entre emisor y receptor "lleno, pero no más": mantener el enlace cuello de botella ocupado transmitiendo, pero evitar altos retardos y formación de fila de espera



$$\text{throughput medido} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Enfoque basado en retardo:

- RTT_{min} - RTT mínimo observado (camino no congestionado)
- `cwnd` con el throughput no congestionado es cwnd/RTT_{min}

if el throughput medido está "muy cerca" del no congestionado
aumentar `cwnd` linealmente /* since path not congested */
else if throughput medido está "por debajo" del congestionado
disminuir `cwnd` linealmente /* since path is congested */

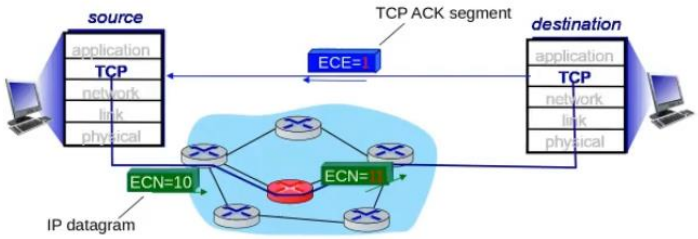
Control de congestión basado en retardo

- control de congestión sin inducir ni forzar las pérdidas
- maximización del throughput ("mantener el 'pipe' lleno... ") mientras se conserva bajo el retardo ("...pero hasta ahí")
- varias versiones de TCP siguen este enfoque
  - BBR (Bottleneck Bandwidth and RTT) se ha implementado en el *backbone* (interno) de Google

Notificación explícita de congestión (ECN)

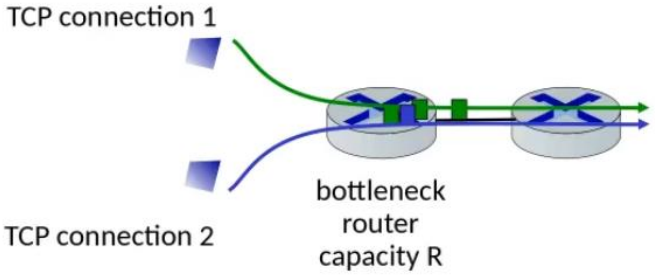
Varias versiones de TCP implementan control de congestión *asistido por la red*:

- Dos bits en el encabezado IP (campo ToS) marcados *por el ruteador* para indicar congestión
  - *política* para determinar el marcaje escogida por el administrador de red
- la indicación de congestión se lleva al destino
- el destino enciende el bit ECE en el ACK para notificar al emisor de la congestión
- involucra tanto a IP (IP header ECN bit marking) como a TCP (TCP header C,E bit marking)



Equidad (*fairness*) en TCP

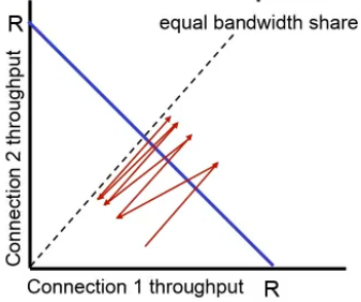
Objetivo de la equidad: si *K* sesiones TCP comparten el mismo enlace cuello de botella de ancho de banda *R*, cada uno debería recibir un promedio de *R/K*



Q: ¿es equitativo TCP?

Ejemplo: dos sesiones TCP compitiendo

- El incremento aditivo da una pendiente de 1, conforme aumenta el throughput
- El decremento multiplicativo disminuye proporcionalmente el throughput



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Equidad: ¿deben ser equitativas todas las aplicaciones?

Fairness y UDP

- Normalmente, las apps multimedia no usan TCP
  - No quieren ver afectada su velocidad por el control de congestión
- Usan UDP en su lugar:
  - Envían audio/vídeo a una velocidad constante, toleran pérdidas
- No hay una "policía de Internet" vigilando que se cumpla el control de congestión

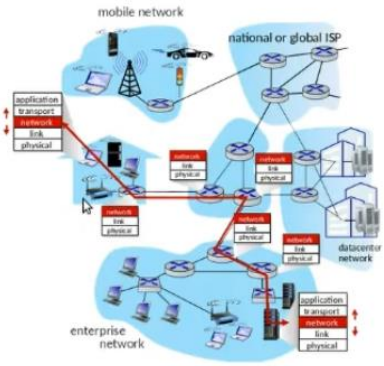
Fairness, conexiones TCP paralelas

- La aplicación puede abrir *varias* conexiones paralelas entre dos estaciones
- Los navegadores web hacen esto, e.g., enlace de velocidad *R* con 9 conexiones:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

14-oct

Servicios y protocolos de la capa de red

- Transporta el segmento desde la estación emisora hasta la receptora
  - Transmisor: encapsula segmentos en datagramas y los pasa a la capa de enlace de datos
  - Receptor: entrega los segmentos al protocolo de transporte
- Hay protocolos de capa de red en todos los dispositivos de Internet: estaciones y ruteadores
- Ruteadores:
  - examinan los campos de los datagramas IP
  - mueven los datagramas de los puertos de entrada a los puertos de salida para transferirlos sobre el camino de extremo a extremo



Funciones clave en la capa de red

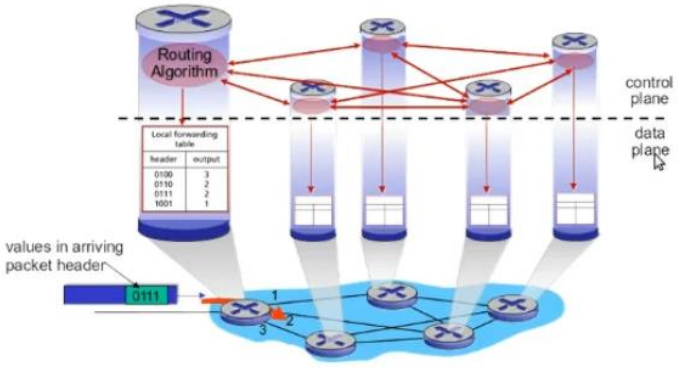
- reexpedición: mover los paquetes desde el enlace de entrada del ruteador hasta el enlace de salida apropiado
- ruteo: determinar el camino que tomarán los paquetes desde la fuente hasta el destino
  - *algoritmos de ruteo*

analogía: darse un paseo

- *reexpedición*: el proceso de atravesar un crucero
- *ruteo*: el proceso de planear un viaje desde el origen hasta el destino

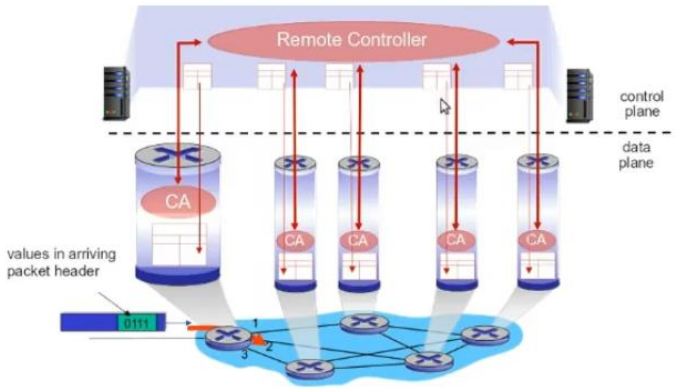
Plano de control *per-router*

Los componentes individuales del algoritmo de ruteo interactúan en todos y cada uno de los ruteadores



Software-defined networking (SDN): plano de datos

El controlador remoto calcula e instala tablas de reexpedición en los ruteadores



Modelo de servicio de red

Q. ¿Qué modelo de servicio escoger para el canal cuando se transportan datagramas desde el emisor hasta el receptor?

Ejemplo de servicios para datagramas *individuales*:

- Entrega garantizada
- Entrega garantizada con menos de 40 ms de retardo

Ejemplo de servicios para un *flujo* de datagramas:

- Entrega ordenada de datagramas
- Ancho de banda mínimo garantizado para el flujo
- Restricciones en cambios en el espaciado entre paquetes

Modelo de servicio de red

Network Architecture	Service Model	Quality of Service (QoS) Guarantees ?			
		Bandwidth	Loss	Order	Timing
Internet	best effort	none	no	no	no

Modelo de servicio *best-effort* de Internet

Sin garantías como:

i. entrega exitosa de datagramas al destino
ii. tiempo u orden de la entrega
iii. ancho de banda disponible al flujo de extremo a extremo

Modelos de servicio de red

Network Architecture	Service Model	Quality of Service (QoS) Guarantees ?			
		Bandwidth	Loss	Order	Timing
Internet	best effort	none	no	no	no
ATM	Constant Bit Rate	Constant rate	yes	yes	yes
ATM	Available Bit Rate	Guaranteed min	no	yes	no
Internet	Intserv Guaranteed (RFC 1633)	yes	yes	yes	yes
Internet	Diffserv (RFC 2475)	possible	possibly	possibly	no

Reflexiones sobre el servicio *best-effort*

- La simplicidad del mecanismo ha permitido la amplia adopción y despliegue de Internet
- La provisión suficiente de ancho de banda permite que el desempeño de aplicaciones en tiempo real (e.g., voz y vídeo interactivos) sea lo suficientemente buena la mayor parte del tiempo.
- Servicios replicados y distribuidos de la capa de aplicación (datacenters, content distribution networks) que están conectados cerca de las redes de los clientes, permiten que se ofrezcan los servicios desde múltiples ubicaciones.
- El control de congestión de servicios "elásticos" ayuda.

Resulta difícil discutir con éxito el modelo del servicio best-effort

Tip

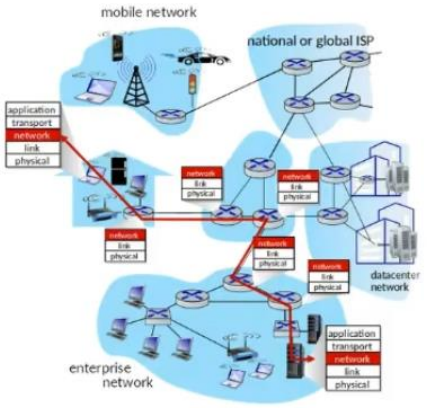
El tráfico elástico no es sensible al retardo.

¿Ejemplo(s)?

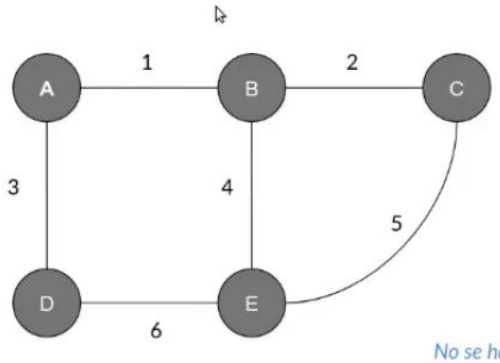
Protocolos de ruteo

Objetivo de un protocolo de ruteo: determinar "buenos" caminos desde el nodo emisor hasta el nodo receptor, a través de una red de ruteadores

- camino: secuencia de ruteadores y enlaces que los paquetes atraviesan desde la fuente inicial hasta el destino final
- "bueno": puede significar menor "costo", más "rápido", menos congestionado
- ¡El problema de ruteo es también un problema en el top 10!



Ruteo por vector distancia



Cold-start

Al inicio del algoritmo, los nodos sólo se conocen a ellos mismos y a los enlaces que tienen.

No se hace diferencia entre estaciones y ruteadores.

Ruteo por vector distancia (cont.)

From A to	Link	Cost
A	local	0

From B to	Link	Cost
B	local	0
A	1	1

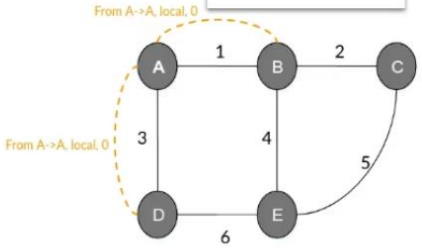
From C to	Link	Cost
C	local	0

Etapa transitoria

Algún nodo "despierta" y difunde, sobre sus enlaces locales, la información que conoce.

From D to	Link	Cost
D	local	0
A	3	1

From E to	Link	Cost
E	local	0



Ruteo por vector distancia (cont.)

From A to	Link	Cost
A	local	0

From B to	Link	Cost
B	local	0
A	1	1

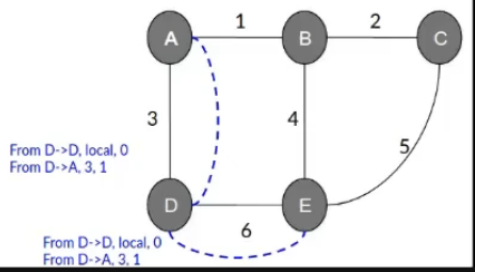
From C to	Link	Cost
C	local	0

Etapa transitoria

Algún nodo "despierta" y difunde la información que conoce sobre sus enlaces locales.

From D to	Link	Cost
D	local	0
A	3	1

From E to	Link	Cost
E	local	0



Ruteo por vector distancia (cont.)

From A to	Link	Cost
A	local	0
D	3	1
B	1	1
C	1	2
E	3	2

From B to	Link	Cost
B	local	0
A	1	1
C	2	1
D	1	2
E	4	1

From C to	Link	Cost
C	local	0
A	2	2
B	2	1
D	5	2
E	5	1

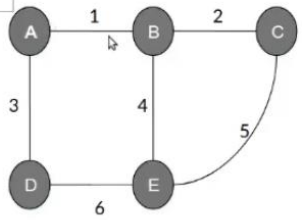
From D to	Link	Cost
D	local	0
A	3	1
B	6	2
C	6	2
E	6	1

From E to	Link	Cost
E	local	0
A	6	2
D	6	1
B	4	1
C	5	1

Etapa transitoria

La difusión de mensajes continúa hasta que las tablas de ruteo ya no cambian.

… y se alcanza el estado estacionario; i.e., el protocolo ha alcanzado la convergencia.



¿Qué pasa si …?

- **Los enlaces son de costos distintos**
  ¿Se llegaria al mismo estado estacionario?
- **Algún enlace falla**
  ¿Es a prueba de fallas el protocolo?
- **El número de nodos es muy grande**
  Ventajas, desventajas … (?)

1.- Seguramente no, porque el objetivo que en las tablas de ruteo haya costos mínimos.

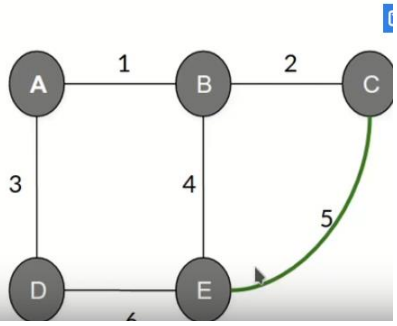
2.- Cada nodo que detecta un cambio o que percibe un cambio en su tabla de ruteo tiene que informales a sus vecinos.

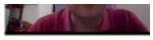
3.- Si tenemos muchos nodos en la red, el protocolo ya no escala, significa que funciona con ciertos números de nodos, pero si se incrementa considerablemente el numero de nodos entonces ya no funciona el protocolo lo que significa que el protocolo no es escalable, un protocolo no escalable significa que no funciona igual a una escala que a otra escala distinta.

Efecto de rebote (*the bouncing effec*

- Los enlaces pueden tener costos diferentes a 1
- Supongamos, costo del enlace 5 = 10, los demás permanecen como al inicio
- Observemos las rutas de todos los nodos a C. En estado estacionario:

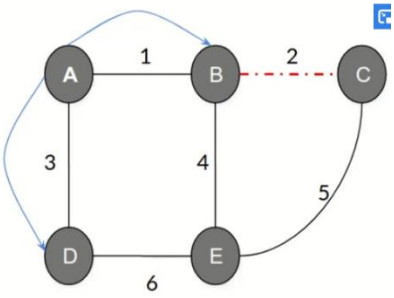
From	Link	Cost
A to C	1	2
B to C	2	1
C to C	local	0
D to C	3	3
E to C	4	2





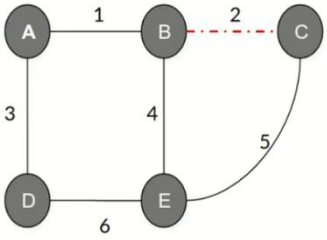
- Ahora, el enlace 2 falla
- B nota el cambio, pero A envía antes su tabla de ruteo a B y D
- Por un instante, la tabla pasa por el siguiente estado

From	Link	Cost
A to C	1	2
B to C	2	∞
C to C	local	0
D to C	3	3
E to C	4	2



- En D no pasa nada
- En cambio, B sumará 1 a la distancia de 2 anunciada por A para llegar a C y observará que 3 < ∞

From	Link	Cost
A to C	1	2
B to C	2	∞
C to C	local	0
D to C	3	3
E to C	4	2



- Ahora, para llegar a C se tiene

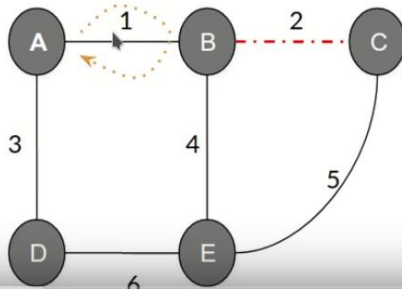
¡Se ha creado un *bucle*!

Todos los mensajes que van a C, llegarán a B y regresarán a A

From	Link	Cost
A to C	1	4
B to C	1	3
C to C	local	0
D to C	3	3
E to C	4	4

TTL: time to live

Se decrementa en la visita de cada ruteador.

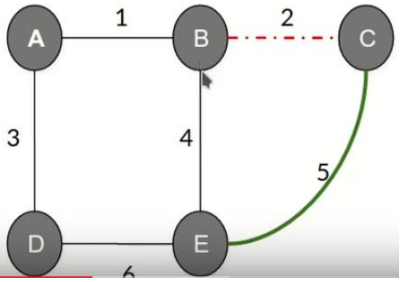


Bucle de ruteo. - Todos los mensajes que estén destinados al nodo C van a pasar del nodo A al B van a dar vueltas, van a estar circulando en la red generando un gran trafico que no es de datos.

El campo TTL.- Es un campo importante porque reduce el tráfico indeseable debido a los bucles

Después de varias iteraciones, se llegará a un nuevo estado estacionario (ver Cap. 4, *Huitema*)

From	Link	Cost
A to C	1	12
B to C	4	11
C to C	local	0
D to C	6	11
E to C	5	10



Counting to infinity and Split Horizon

Counting to infinity

La red podría desconectarse si, por ejemplo, dos tramos fallan

Se detiene el bucle creado con la representación de infinito.

Split Horizon

Otra técnica para minimizar el efecto de rebote en el protocolo vector distancia

Revisar *Triggered Updates*

El protocolo RIP (RFC 2453) es el mejor ejemplo de un protocolo de vector distancia

¿Cuál es el período de emisión de mensajes de *refresco*?